A New Perspective for Efficient Virtual-Cache Coherence

Stefanos Kaxiras
Department of Information Technology
Uppsala University, Sweden
stefanos.kaxiras@it.uu.se

Alberto Ros
Department of Computer Engineering
University of Murcia, Spain
aros@ditec.um.es

ABSTRACT

Coherent shared virtual memory (cSVM) is highly coveted for heterogeneous architectures as it will simplify programming across different cores and manycore accelerators. In this context, virtual L1 caches can be used to great advantage, e.g., saving energy consumption by eliminating address translation for hits. Unfortunately, multicore virtual-cache coherence is complex and costly because it requires reverse translation for any coherence request directed towards a virtual L1. The reason is the ambiguity of the virtual address due to the possibility of synonyms. In this paper, we take a radically different approach than all prior work which is focused on reverse translation. We examine the problem from the perspective of the coherence protocol. We show that if a coherence protocol adheres to certain conditions, it operates effortlessly with virtual caches, without requiring reverse translations even in the presence of synonyms. We show that these conditions hold in a new class of simple and efficient request-response protocols that use both self-invalidation and self-downgrade.This results in a new solution for virtual-cache coherence, significantly less complex and more efficient than prior proposals. We study design choices for TLB placement under our proposal and compare them against those under a directory-MESI protocol. Our approach allows for choices that are particularly effective as for example combining all per-core TLBs in a single logical TLB in front of the last level cache. Significant area, energy, and performance benefits ensue as a result of simplifying the entire multicore memory organization.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Parallel processors*; B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms

Design, Experimentation, Performance

ISCA '13 Tel-Aviv, Israel

Keywords

Multicore, virtual caches, cache coherence, request-response protocol, self-invalidation, synonyms, TLB organization

1. INTRODUCTION AND MOTIVATION

While researchers have long advocated the benefits of virtual caches [7, 8, 14], virtual-cache coherence remains particularly complex. Few, if any, contemporary architectures implement coherency for virtual caches. However, in today's power-constrained multicore architectures, coherent virtual caches can become a key element for power-efficient implementations, if we can overcome their complexity. This paper proposes a new approach towards this goal.

Coherence, by definition, deals with a single memory location (memory block). Because a virtual address of a block cannot be used as a single point of reference in the whole system for that block, ultimately coherence must be based on the block's unique physical address. In practice, the transition from the virtual address space to the physical address space, occurs even before the tag match in the L1; physical addresses are used onwards. Unfortunately, this means that TLB energy is consumed on every L1 access. Recent work by Basu *et al.* [2] expounds on the power benefits of avoiding address translation for a single (fat) core using virtual caches. But the benefits are even more pronounced in a manycore with many thin cores, where placing a TLB alongside each core and its L1, incurs relatively more area and power cost.

With virtual caches the TLB is moved after the L1. Energy is saved on L1 hits and expended only on misses. But this comes at the price of a significant increase in complexity (and overhead) in enforcing coherence [31]. Reverse translation from physical to virtual addresses is now required for coherence traffic sourced at a physically-addressed directory and directed towards a virtual L1 (e.g., invalidations, downgrades, forwardings). The reason is that the virtual address under which the data reside in the target virtual L1 is not always known. Even if one considers carrying both a virtual address and its physical-address translation with every coherence request (a significant overhead in its own), reverse translations are still needed because of the possibility of synonyms, i.e., different virtual addresses mapped to the same physical location. One can ban synonyms at the software/OS level and circumvent these problems. But synonyms are *sine qua non* for modern software development. Solutions that constitute reverse translation, are dual physical-address tags in the L1s [2, 14], or private physical L2s with backpointers to the L1s [34]. Others aim to es-

tablish, via OS involvement and centralized mapping structures, a unique primary virtual address [28] or a unique intermediate-address representation [39].

The contribution of our work is a simple while efficient solution intended for multicore cSVM that: i) supports synonyms, ii) requires no reverse translations, and iii) yields significant benefits in energy, performance, and area, while at the same time considerably simplifying coherence.

We make the following observation: *Virtual-cache coherence (supporting synonyms) without reverse translations is possible with a protocol that does not have any request traffic directed towards virtual L1s; in other words, a protocol without invalidations, downgrades, or forwardings, towards the L1s.* It is actually possible to maintain coherence in an efficient manner, even under these stringent constraints. Self-invalidating protocols are long known to eliminate the first kind of traffic towards the L1 (invalidations), requiring only data-race-free (DRF) semantics for ordinary data [22, 11, 19]. In our prior work, we proposed the VIPS-M protocol that also manages to eliminate downgrades and forwardings [29]. VIPS-M, in addition to self-invalidation, uses *self-downgrade* via a delayed-write-through policy for the shared data in the L1. In this work, we use this protocol because it is application-transparent, simple, power- and performance-efficient. We note however, that any other protocol with similar properties can be used in its place, and we identify alternative candidates.

Today, the value of virtual-cache coherence is amplified by an emerging class of heterogeneous architectures that would greatly benefit from it. These are architectures where GPUs or manycore accelerators are coupled with general purpose cores on the same chip. In this context, virtual-cache coherence can be the key for implementing coherent Shared Virtual Memory (cSVM) in an efficient manner, saving a significant part of the address translation in the accelerator or general-purpose cores. cSVM is, of course, highly desirable in a heterogeneous architecture because it provides a uniform view of memory from any core. For example, using a single virtual address space, GPU cores can safely access arbitrary pointer-based data structures, created in a general purpose core. This is the direction rigorously pursued by industry today [6, 13, 33, 36].

To summarize, we provide a new solution for one of the fundamental problems in memory hierarchy design, how to efficiently implement virtual-cache coherence, by examining the problem from an entirely new point of view: that of the coherence protocol itself.

- We lay out the conditions under which virtual-cache coherence with support for synonyms, does not require reverse translations or any equivalent mechanisms. (§ 3)
- We show how a new class of very simple but efficient cache coherence protocols, without any extra hardware, is sufficient to implement virtual-cache coherence. We show how synonyms are handled in these protocols. (§ 4 and § 5)
- Having this novel solution for virtual-cache coherence, we explore design choices for the placement and the sharing of TLBs in a multicore architecture. We compare against a MESI directory protocol that requires reverse translations and show that our solution yields significant benefits and efficient TLB configurations. (§ 6, § 7, and § 8)
- We show that, using our solution, a multicore can be serviced even by a single (logical) TLB at the LLC. In this configuration, there is no need to keep multiple TLBs (one per core) coherent, which entails extra complexity [32]. Further, the use of TLB storage is maximized because there is no replication of TLB entries. (§ 6, § 7, and § 8)

2. BACKGROUND

Implementing coherence for virtual caches is difficult because: i) Virtual caches must be accessed by virtual addresses, while coherence ultimately must use a unique physical address as a single point of reference. This implies the need for both forward and reverse translations for typical coherence protocols. ii) The problem of synonyms complicates coherence with potentially multiple results per reverse translation. This section reviews prior solutions to the synonym problem in virtual caches, with and without coherence.

2.1 Synonym problem in a single cache

The synonym problem plagues virtual caches, where no address translation is performed prior to accessing them. Thus synonyms allow for the possibility of multiple copies of the same data co-existing in the cache. There are two general approaches to deal with synonyms in a virtual cache, described in detail by Cekleov and Dubois [7]: i) prevent or avoid synonyms in software (see [7] for a summary) or ii) detect and manage synonyms in hardware, which we summarize below.

The tenet of synonym management based on hardware detection can be summed up to a simple rule: *when multiple synonyms are detected in a cache, allow only one to exist.* Synonym detection is performed on every miss, thus guaranteeing that multiple copies of the same data cannot exist simultaneously in the same cache. Two basic approaches are:

- Brute-force search. The cache controller, on any miss, searches all possible sets in the cache where a synonym could reside. Brute force search is acceptable if the cache is virtually-indexed but physically-tagged (VIPT) but rather painful for a virtually-indexed and virtually-tagged cache (VIVT), where each virtual tag in the searched sets must be translated separately at the TLB.

- Reverse maps [14] or L2 backpointers [34]. These structures map a physical address to a cache line in a virtual cache. If, on a miss, a " backpointer" is found in the reverse map or the L2, for a particular physical address, then it points to a synonym in the L1 cache.

Regardless of the detection technique, the management of the synonyms is the same. If on a miss, a synonym is detected, it is either evicted or re-tagged (and re-indexed if necessary) under the missing virtual address.

2.2 Synonym problem in coherence

The synonym problem also affects coherence in a multiprocessor (e.g., multicore) where synonyms can exist in different virtual caches. Coherence operations must now deal with multiple virtual addresses for the same block.

2.2.1 SPUR

SPUR implemented virtual caches on a shared bus [16]. Address translation is performed in software after a miss. Coherence on the memory side is maintained on physical addresses and on the cache side on virtual addresses. The shared bus carried both addresses simultaneously. Recognizing the complexity of virtual coherence, SPUR disallowed synonyms altogether, so blocks are identified by a unique virtual address.

2.2.2 Goodman's solution

Goodman proposed one of the earliest solutions for virtual-cache coherence allowing synonyms [14]. In his proposal, virtual caches include both virtual tags accessed by the core (V-tag memory) and physical-tags (R-tag memory) accessed by the coherence protocol. R-tag memory is a reverse map that contains backpointers both to the V-tag memory and the data array. The V-tag memory, and R-tag memory, can have different organizations, leading to complex eviction cases where two lines may be evicted at once, in order to secure a compatible pair of V-tag and R-tag entries. Coherence takes place entirely in the physical address space through the R-tag memories, which perform a reverse translation from physical addresses to virtual addresses. Goodman's proposal is representative of other similar solutions, e.g., the software controlled coherence in VMP [10], and carries a significant price in terms of complexity and overhead (dual virtual-physical tags).

2.2.3 Private physical L2s

Another design point proposed by Wang *et al.* is to back virtual L1 caches with *private* physical L2s (as in the single-cache case discussed above) [34]. This easily solves the coherence problem, in the presence of synonyms, since coherence among the L2s can now be enforced on unique physical addresses. Backpointers to the L1 are required in each L2 to prevent more than one synonym per line to exist in the L1 and inclusion must be enforced. In essence, as Wang *et al.* explain, the entire private L2 becomes an expensive reverse translation mechanism for its L1 [34]. Unfortunately, the physical L2s cannot be shared since this would return us back to square one with respect to the coherence of the virtual-L1s. This fundamental limitation renders this approach impractical in the situations where virtual-cache coherence is desired the most: in the GPUs or manycore accelerators of a heterogeneous architecture where many thin cores require efficient coherence but cannot afford the cost of private L2s.

2.2.4 Other solutions

Finally, some more recent proposals make strong cases for virtual caches. Qiu and Dubois propose a technique called the synonym lookaside buffer (SLB) that aims to establish a unique (primary) virtual address among synonyms [28]. The primary address, being unique for a block, is used to enforce coherence in virtual addresses. However, their technique relies on substantial involvement of the operating system to identify and maintain a primary address for each page [28]. Similarly, Zhang *et al.* propose using another level of indirection in the address translation to sidestep the synonym problem: unique intermediate addresses [39]. Again, substantial involvement from the OS is required and large centralized structures manage the translation from virtual to intermediate to physical [39]. Basu *et al.* propose another approach called opportunistic virtual caching (OVC) [2]. OVC exploits the rarity of synonym accesses —also reflected in our experience— and with appropriate OS and user support, selectively switches between virtual caching and physical caching. Coherence is maintained by serially searching physical tags (which are available in the virtual caches) for all sets that might hold a block. Thus, OVC is a combination of a reverse map (embedded in the tags) and a brute force synonym search.

We view the problem from an entirely different perspective: that of the coherence protocol itself. We offer a new solution (unlike any prior) to the problem of virtual-cache coherence, that is both simple and effective.

3. WHY IS REVERSE TRANSLATION USED?

As we have seen in prior solutions, reverse translations in one form or another (reverse maps, L2 backpointers, etc) are used to access virtual caches from the coherence side (which is entirely in physical addresses). Unfortunately, reverse translation introduces significant complexity and overhead. For us, the root of the problem lies in the coherence protocols themselves.

While there are many definitions of coherence, the Single-Writer-Multiple-Readers invariant, put forth by Sorin *et al.* [31] gives an intuitive sense of what a typical cache protocol does. It boils down to two fundamental operations:

- Upon a write, the cache coherence protocol must find all the (read) copies of the data and invalidate them.
- On a subsequent read, the cache coherence protocol must provide the latest value of the data by locating the last writer; the last writer is downgraded to a reader.

These two operations are straightforward if all the cache copies of the data (readers and writer) are identified by their unique physical address in a directory indexed by physical address. Alternatively, in snooping solutions, the physical address of the reads and writes is broadcast to all caches.

However, the same operations are problematic in virtual addresses.[1] Assume, for example, virtually-indexed virtually-tagged (VIVT) L1 caches. Coherence requests from the virtual caches (reads or writes) undergo address translation via a TLB before they reach the directory. According to the two fundamental operations of a cache coherence protocol:

- A write request reaching the directory can generate a number of invalidation requests. Each of these new requests, requires its own reverse translation because of the possibility that in the target cache the data exist under a different virtual address than the one used by the write request. Worse, if multiple synonyms are allowed to exist in the same cache, a single invalidation request may result in multiple translations to virtual addresses.
- A read request that reaches the directory is forwarded to the last writer of the data that is tracked by the directory. This indirection also requires a reverse translation as the writer may use a different synonym than the reader.

[1]Obviously, if data are identified by a unique system-wide virtual address (or some other unique representation [28, 39] —given extensive OS and hardware support) then the coherence mechanisms described above are not affected.

From this discussion we conclude that a necessary condition to avoid reverse translation *in the coherence protocol* is the following: *All coherence request traffic should be strictly one-way from the virtual address domain to the physical address domain —and never from physical to virtual, or virtual to virtual.* In a MESI-like protocol, invalidations, forwardings and downgrades violate this condition.

4. THE RIGHT STUFF

While the aforementioned conditions impose significant restrictions, in fact, coherence protocols with the right properties do exist. One has to look into very simple request-response protocols [29] or even simple, GPU-specific coherence [30]. Excluding purely software-driven coherence (that puts the responsibility of maintaining coherence entirely on the program), we have identified a number of potential candidates:

- The Dir_1SW protocol proposed by Wood *et al.* [38] is a potential candidate. In this protocol a simple directory tracks either a single writer or the number of readers. Any operation that requires forwarding or multiple messages (e.g., invalidations, acknowledgments, etc.), traps to software. Handling complex coherence operations in software (where the information on synonyms is available) allows one to circumvent reverse translation mechanisms in hardware, but the overhead is likely to be noticeable.
- Better than the Dir_1SW (for virtual-cache coherence), is the cooperative shared memory approach, (or Check-In/Check-Out —CICO— model, as it is also known), proposed by Hill *et al.* [17]. Cooperative shared memory builds upon Dir_1SW but relies on program annotations to eliminate software traps. In the CICO model, program annotations in the form of the Check-in and Check-out directives, effectively implement program-driven self-invalidation/self-downgrade, i.e., the right ingredients to avoid reverse translations (even as software traps). However, program annotations are optional, and cooperative shared memory degrades to Dir_1SW without them.
- Lastly, we recently proposed a protocol called VIPS-M, that satisfies the above condition [29]. We have chosen to illustrate our approach to virtual-cache coherence using this protocol, because it is efficient, easy to implement, and application-transparent. VIPS-M implements self-invalidation and self-downgrade based on synchronization and data-race-free (DRF) semantics. Because many researchers champion DRF semantics in parallel programming (see for example the work of Choi *et al.* [11] among others), we believe that this is a reasonable condition for enabling a significant reduction in the complexity and cost of (physical or virtual) coherence.

4.1 VIPS-M

VIPS-M is a directoryless protocol that uses both self-invalidation and self-downgrade and as a result eliminates all request traffic towards L1s. Self-downgrade is implemented as a write-through policy. Whereas a pure write through policy is widely known to be damaging to performance, the key observation of this work is that most of the write misses in a write through policy (over 90%) are due to private data. Write through is only needed for data that are actually shared —not for private data that do not need coherence and can still follow a write back policy. As with other efforts to simplify coherence [11, 19], VIPS-M relies on the properties of relaxed memory consistency and data-race-free semantics to allow incoherence in between synchronization points, but sequential consistency for data-race-free programs.

The VIPS-M protocol is based on three mechanisms: i) a data classification mechanism that classifies data (cache lines) as private or shared, ii) self-downgrade, and iii) self-invalidate.

4.1.1 Private/shared data classification

Data classification in VIPS-M is based on a widely used technique that classifies data at a page granularity using the OS and the TLBs [12, 15, 20]. The advantage of an OS technique is that it does not impose any extra requirements for dedicated hardware, since private/shared information is stored along with the page table entries.

In the OS technique, a page accessed by a single core starts as private in the page table, but when a second core accesses the same page, it becomes shared. When a core notices that a page is tagged earlier as private by another core, the latter needs to be interrupted and its TLB entry updated so it can see the page, henceforth, as shared. While this may be an expensive operation, it is rather rare. Unfortunately, if a single block in the page is shared (or even if two different private blocks within the same page are accessed by different cores) the whole page must be considered as shared, thus, leading to misclassified cache lines.

4.1.2 Self-downgrade

The VIPS class of protocols introduces an efficient form of self-downgrade, whereupon written data are deliberately put back in the LLC before any further access to them by other cores. This obviously eliminates forwarding and cache-to-cache transfers, present in other protocols. The straightforward implementation of a self-downgrade policy is with a simple two-state (Valid/Invalid) write-through policy in the L1s. The efficiency comes from data classification into private and shared which results in a dynamic write policy in the L1s:

- Write-back for private cache lines that do not need coherence.
- Write-through for shared cache lines.

Even with this distinction, write-through still generates a significant amount of traffic, thus VIPS-M implements a delayed write-through, using the miss-status holding registers (MSHRs). This reduces the amount of write-throughs by coalescing multiple writes, and brings the total traffic close to that of a write-back policy for all data. The "delayed-write-through" state, where the cache line can be written repeatedly without any further action, exists only from the write miss to the write-through and is invisible to transactions from other cores —therefore introduces no complexity to the protocol.

The delayed write-through is implemented in the core's MSHRs. Each MSHR is associated with a timer causing the write-through to occur at a fixed delay after the initial write. Write-throughs are forced to complete if an MSHR entry is replaced or if all the MHSRs are flushed at synchronization.

Because data-race-free semantics are mandated only for word (or even byte) level and not on the whole cache line,

false sharing (e.g., two writers wring on different words of a cache line at the same time) can have adverse effects. For this reason, write-throughs transfer only what is modified in a cache line (i.e., a diff). Multiple simultaneous writers are allowed to co-exist as long as they are data-race-free at word(byte) level. Because diffs are created from the time of a write miss to the actual write through, dirty bits per word(byte) are only needed in the MHSRs, resulting in an efficient implementation [29].

4.1.3 Self-invalidation

Finally, VIPS-M eliminates directory invalidation using self-invalidation [22]. Readers, obtain a "tear-off" copy of block, and do not need to be invalidated by writes, as long as they self-invalidate their copy, on their own, at (or before) the next synchronization point they encounter. In VIPS only the shared data are invalidated (based on the data classification). Further, the same OS classification technique described previously distinguishes between "Read-Only" pages (pages that have not been written) and pages that are Read-Write. Cache lines belonging to Read-Only pages are not self-invalidated at synchronization.

The selective self-invalidation step eliminates the need for a directory, since neither the writers (because of self-downgrade) nor the readers (because of self-invalidation) need to be tracked anymore. Shared (data-race-free) data are handled without any state or blocking in the LLC. Any miss, whether private or shared, read or write, is satisfied with a simple request-response to the LLC. The only difference between private and shared data is in when they are put back in the LLC.

In our prior work we also discuss how VIPS-M handles *intentional* data races such as those caused by concurrent, atomic, read-modify-write instructions intended for synchronization [29]. Such instructions invoke a different protocol, that bypasses the L1 cache. This solves the problem of not having invalidations to signal changes to cores that are spinning. While their "synchronization" protocol goes directly to the LLC, in many cases (e.g., in small critical sections) spinning is reduced due to delayed write-throughs.

5. SYNONYM HANDLING IN VIPS-M

It is clear that VIPS-M has some properties that set it apart from typical coherence protocols:

- It is strictly a request-response protocol from the L1s to the LLC. There is no other coherence traffic.
- It is truly distributed, i.e., coherence decisions are taken independently, without any interaction among cores.

The first property is sufficient to satisfy the condition put forth in § 3 for eliminating reverse-translation. The key for synonym coherence, however, is the second property: *truly distributed* coherence that guarantees that invalidation and correct update of an L1 copy occurs irrespective of (its) synonyms.

Synonyms are commonly used to implement either shared-memory or message-passing semantics. Such synonyms are accessed in parallel following DRF semantics (i.e., conflicting accesses are separated by synchronization) and VIPS-M unequivocally keeps them coherent. In situations with active sharing, this type of synonym is not only the dominant type, but also the most performance-critical. It is precisely in this case where our approach is most appealing due to its simplicity and effectiveness.

Synonyms, however, can also be created as a side effect of virtual memory management (e.g., virtual page demappings and remappings, or physical page deallocation and reallocation in demand paging). As these synonyms are not created frequently [2], nor accessed in parallel [28], nor used for communication, their coherence is not a performance issue but rather a correctness issue; thus, they are preferably delegated to the operating system [7] rather than complicating the underlying hardware. Typically, operating systems provide ample support for these synonyms (e.g., [9]). Their handling is explained below.

5.1 Synonyms for shared-memory semantics

Self-invalidation in VIPS-M implies a weak memory consistency model [22, 11, 29]. In such a model, accesses to ordinary (non-synchronization) shared data follow data-race-free semantics. This guarantees that when a synonym is written, its latest value will be propagated to the LLC (via a delayed write-through) before the next synchronization by the writer thread. Similarly, to see a new value, a reader thread will have to cross a synchronization point, whereupon it will self-invalidate its synonym (as any other shared data). If data races do exist for synonyms (e.g., for synchronization variables), the VIPS-M synchronization protocol is used instead [29], making sure that the L1 copy is automatically self-invalidated and the LLC copy is re-read.

A unique characteristic of our approach is that we allow synonyms to *simultaneously exist in the same cache*, as VIPS-M handles their coherence irrespective of their location.

5.2 Synonyms for message-passing semantics

Another common occurrence of synonyms is for optimizing message-passing semantics to avoid physical memory copies. This is done by remapping physical pages from the sender to the receiver. Because these synonyms are used for sharing data (messages), albeit not with overlapping accesses, their coherence is still performance-critical. Thus, they are best handled by VIPS-M rather than by the OS. VIPS-M is invoked by classifying synonym pages as Shared. DRF semantics are imposed by message-passing semantics. VIPS-M provides an efficiency in handling these synonyms that is impossible to achieve with the OS. To enforce coherence, the OS would *flush* the page from the cache. Flushing necessitates writing back all dirty data in a burst, and this can be a serious bottleneck. In VIPS-M, writes are paced (over time) to the LLC with the delayed-write-through policy, thus avoiding network saturation. On synchronization, shared data need only be *purged* (invalidated) —a lightweight and fast operation on the valid bits. Only outstanding delayed-write-throughs resident in the MSHRs need to complete, thus keeping the overhead minimal.

5.3 Aliases

There is also the possibility that synonyms are created not as a result of sharing data, but for other reasons. Aliases[2] are due to remapping a physical page (P) from one virtual page (V1) to another (V2), either because of demapping V1 or swapping V1 to the disk and subsequently mapping P to

[2]We follow the naming convention in [7, 8]

V2 —in either case, there is no data sharing involved. Although the mappings V1→P and V2→P do not exist concurrently, aliases, if left in the cache, can cause errors by giving access to data via stale mappings. Since aliases are not accessed following DRF semantics, VIPS-M, by itself, cannot keep them coherent. However, there is no need for that, since coherence for aliases is just a correctness issue —*not a performance concern*. Clearly, the frequency of a page demap or a page swap is not very high. The OS simply *flushes* V1 from the cache before the demapping or the swapping occurs, to clear latent blocks that still use the defunct mapping. In this way, page P remains classified as private throughout.

5.4 User-requested, same address-space synonyms

Finally, some systems allow user-requested synonyms to be established in the same context (address space). Previous work shows that this type of synonym is exceedingly rare [2, 35] and even argues that it is unnecessary for programmers [35] (some systems disallow this type altogether [21]). Although, we do not encounter these synonyms in our workload, we discuss their handling for completeness.

We allow these synonyms (with VIPS-M coherence for performance) either as read-only, or as DRF-compliant. The latter means that conflicting accesses must be separated by "synchronization events" (i.e., forcing self-invalidation and completion of the delayed-write-through) even if the conflicting accesses belong to a single thread of execution. If, say, for the sake of backwards compatibility, we would need to handle such synonyms but without having a guarantee for DRF-compliance, we resort to the Mach OS approach [9]: synonym pages are set to read-only and writes trap to the OS where they are resolved.

5.5 Private/shared classification for synonyms

In VIPS-M, pages accessed by more than one core are classified as Shared [29], and coherence is enforced on their cached data. However, this is inadequate for inter-process synonyms that involve virtual pages in more than one page table. Given our discussion above for the different types of synonyms, we note, that the OS is aware of when it creates a synonym and of what type. The OS then classifies as Shared (by default), synonym pages at the moment of their mapping, if they are:

- virtual pages that map on the same physical page with mappings that overlap in time (shared-memory semantics),
- virtual pages that map sequentially on the same physical page, but *preserve* the data in the physical page between mappings (message-passing semantics),
- synonyms in the same address space (intra-process, DRF-compliant).

All other pages start as Private and are subject to the normal classification technique [29].

6. DESIGN CHOICES FOR TLB PLACEMENT

Having a simple solution for virtual cache coherence allows us to consider various options for TLB placement. We analyze three design decisions based on different TLB placements: Core-TLB-L1-Network-LLC (or cT1NL) , Core-L1-TLB-Network-LLC (or c1TNL), and Core-L1-Network-TLB-LLC (or c1NTL). The last one is a new option for virtually-indexed virtually-tagged (VIVT) caches that is especially appealing with our proposal. The three configurations are depicted in Figure 1. Details about behavior, drawbacks, and advantages of each configuration both for MESI and for VIPS-M protocols are given in the following sections.

6.1 cT1NL: Physically-tagged L1 caches

In this classical organization first-level caches are physically tagged. As we show in Figure 2a, in the simplest form, the TLB is accessed first, to get the physical address of the requested block, with which to access the L1 and perform the tag comparison. Although this organization is simple, it has the following drawbacks. First, accessing the TLB on every memory reference leads to a high energy consumption. Second, the TLB access is in the critical path of every L1 cache access.

To achieve faster L1 access time, usually the TLB access is overlapped with the L1 access by indexing the L1 with the virtual address (i.e., a VIPT cache). To avoid synonyms occupying different sets and allow coherence in physical addresses, the virtual and physical indexing must be the same. This restricts virtual indexing to the page offset bits, thus limiting the size of the L1 to one page per way [18].[3] The size of the TLB is also restricted by its access latency, so that cache tag comparison is not delayed.

6.1.1 MESI operation

In case of a cache miss, a new entry storing the physical address of the requested block is allocated in the miss status holding register (MSHR) structure. This entry keeps track of the coherence transaction. Subsequently, a request message carrying the physical address and the MSHR index (4 bits in our particular implementation since we have 16 MSHR entries) is sent to the LLC through the network. The LLC looks for the data block, and in case of indirection (e.g., the block is in M state in another cache), the request is forwarded to the cache holding the block. At this cache, the data block is obtained and sent to the requesting cache. As an optimization, this response does not carry the physical address but just the MSHR index, as done in the AMD's coherent HyperTransport (cHT) protocol [26]. When the data response arrives to the requesting cache, it accesses the MSHR to obtain the physical address and to check that the miss has been resolved (e.g., all acknowledgements of invalidations have been received in case of a write miss).

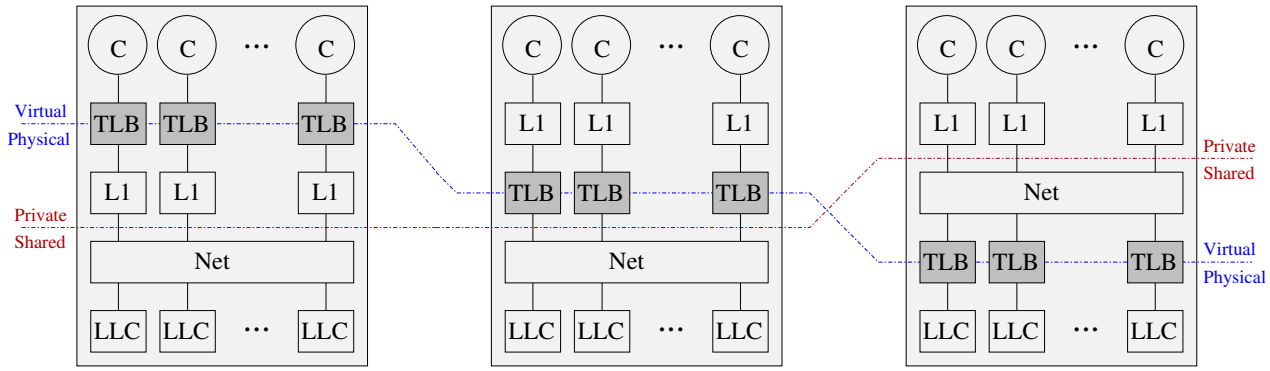
6.1.2 VIPS-M operation

The VIPS-M protocol has a similar behaviour, as depicted in Figure 3a, but it never suffers from indirection, since the data block will be always available at the LLC (or in main memory).

6.2 c1TNL: Virtual L1 caches, private TLBs

L1 virtual caches can largely reduce the energy consumed by address translation, since TLBs only need to be accessed upon cache misses. Additionally, since the TLB is not accessed in the critical path of L1 cache accesses, its size can

[3]Or, requires that some low order bits in the virtual page number be the same as in the physical page number (thus placing restrictions on page allocation).



(a) cT1NL (b) c1TNL (c) c1NTL

Figure 1: Systems configurations evaluated

be larger, thus being able to achieve a higher hit rate. While the TLBs' LRU replacement policy may underperform due to L1 filtering, the benefit is still significant. Note that, cache hits are possible even if the TLB entry for that page is not present because of a previous eviction. Finally, restrictions in the number of cache sets of the L1 cache due to the virtual indexing are lifted.

On the other hand, virtual caches introduce the synonym problem, as discussed in the previous sections. Extra fields, such as the process ID (P_{id}) and permission bits, should be also added to the cache to handle homonyms (identical virtual addresses in separate address spaces, mapped to different physical addresses). For multiprocessors, virtual caches also require reverse translations (from physical to virtual).

Cache coherence protocols have to deal with these problems [8]. First, since synonyms within the same cache are not desired [7], a check of possible cached synonyms for the requested block must be performed on a cache miss. Second, upon arrival of a coherence transaction (i.e., invalidation, downgrade, or forwarding) as a consequence of a miss in another cache, a reverse translation is needed to find the block in the virtual cache. A common way of doing this is by adding an R-tag memory [14] able to perform physical-to-virtual translations.

6.2.1 Reverse translation

The R-tag memory is a physically-indexed and physically-tagged (PIPT) cache that stores the virtual translation for any block in the virtual L1 cache (or alternatively a pointer to the corresponding virtual cache line) [14]. The physically-indexed R-tag memory must provide an entry for each and every block in the virtually-indexed L1. This is easily achieved with identical indexing and associativity. However, only the page offset bits are the same in the physical and the virtual address and the L1 virtual index can expand beyond the page offset bits (since the size of the L1 ways is not constrained). In this case, the R-tag physical index can only be a subset of the L1 virtual index, limited to page offset bits (thus constraining the number of R-tag sets to the number of blocks in a page). The difference in index size is made up in associativity (i.e., if the virtual index is n bits larger than the physical R-tag index then R-tag associativity is 2^n times larger than L1 associativity). In contrast, with incompatible virtual and physical indexing, multiple cache evictions could be required when checking for synonyms [14, 7] leading to a significant increase in complexity.

6.2.2 MESI operation

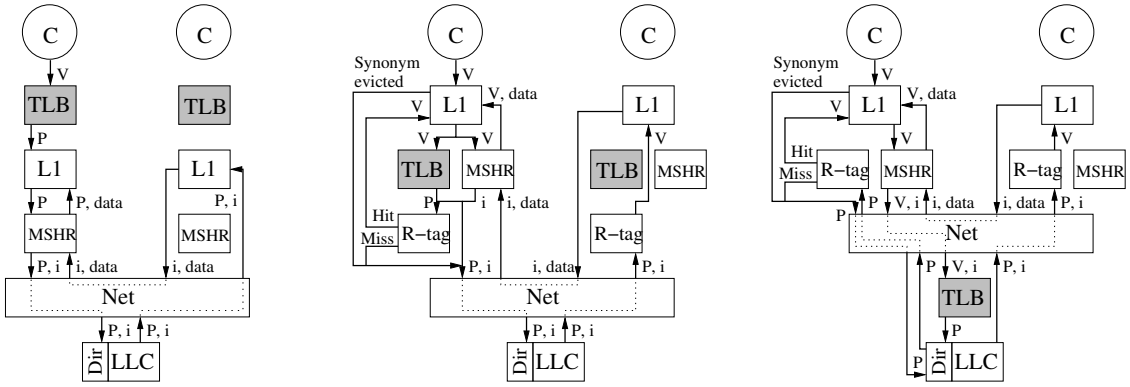
A cache miss for a MESI protocol with virtual caches works as depicted in Figure 2b. First, the TLB is accessed in order to get the physical address of the missing block. Then, the R-tag memory is accessed. In case of a hit in the R-tag, a synonym exists and must be evicted. In parallel with the TLB access, a new entry storing the *virtual address* of the missing block is allocated in the core's MSHRs. Once potential synonyms have been evicted, a request carrying the physical address and the MSHR index is issued to the LLC. In case of indirection, the request is forwarded to the cache holding the block and the local R-tag is accessed to locate the block in the cache. The block is returned to the requesting core along with the MSHR index. At the requesting core, the corresponding MSHR returns the virtual address of the miss, allocates the data block in the virtual cache, and completes the coherence transaction.

6.2.3 VIPS-M operation

In VIPS-M, synonyms in the same cache are handled as described in § 5. Additionally, there are no invalidations, downgrades, or forwardings to third-party caches. Therefore, cache misses are resolved in a simpler way without an R-tag memory (see Figure 3b) which saves latency, power, and area. More specifically, upon a cache miss the TLB returns the physical address, and, in parallel, a new entry is allocated in the MSHRs (storing the *virtual* address). A request with both the physical address and the MSHR index is sent to the LLC across the network. The LLC sends back the data block with a response that carries just the MSHR index, as in [26]. When the requesting core receives the data message, the corresponding MSHR returns the virtual address and the received data are stored in the cache.

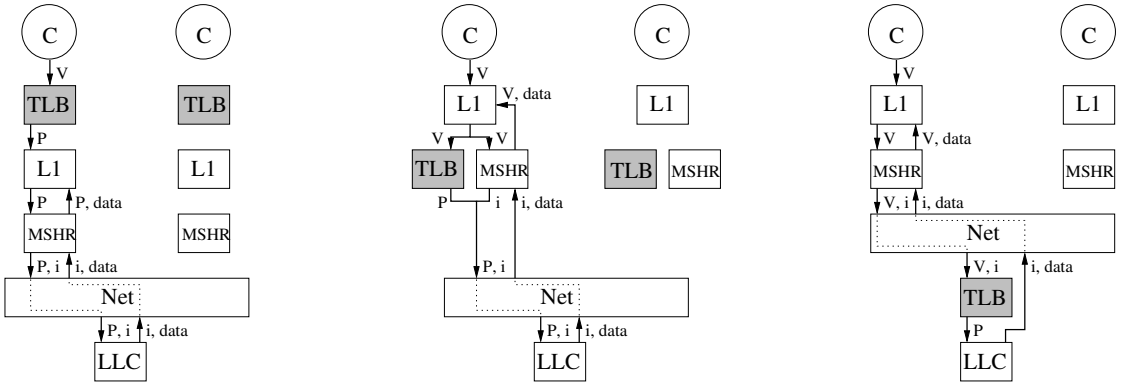
6.3 c1NTL: Virtual L1 caches, shared TLBs

Since in the two previous configurations TLBs are private, they need to be kept coherent [32]. TLB coherency complicates system design even more, and can lead to an increased number of TLB misses [5]. Also, private TLBs may replicate information (same mappings in many TLBs), thus lowering their aggregate effective capacity. However, with



(a) MESI-cT1nL (b) MESI-c1TnL (c) MESI-c1nTL

Figure 2: MESI operation examples. V: virtual address; P: physical address; i: MSHR index; data: requested data block.



(a) VIPS-M-cT1nL (b) VIPS-M-c1TnL (c) VIPS-M-c1nTL

Figure 3: VIPS-M operation examples. V: virtual address; P: physical address; i: MSHR index; data: requested data block.

our proposal a new possibility opens up: moving the TLB *after* the network (just before the LLC access) where it can be logically *shared*. Bhattacharjee, Lustig, and Martonosi have already demonstrated significant benefits of sharing the second-level TLBs [3]; we propose this for a single TLB level. At this location, the TLB is far-removed from the cores and issues arise concerning handling of memory traps. However, Qiu and Dubois have already addressed such problems in their work [27].

6.3.1 MESI operation

Traditional solutions for virtual caches [14, 34], cannot support efficiently a shared TLB. Wang *et al.* proposal [34] is incompatible with this solution since it relies on a physically-indexed L2 private cache, so the TLB must be placed between the L1 and the L2, and not between the network and the LLC. Goodman's proposal [14] has the problem of not been able to access the R-tag memory just after the cache miss, since in the c1nTl configuration the translation is performed after crossing the network. A naïve solution is to perform the translation after the network and then send a message back to the R-tag memory to check for synonyms. This, however, can be optimized by checking the directory first. If the directory indicates that the physical address is present in the same L1 cache generating the miss then a synonym can potentially exist (false positives can occur due to silent evictions). In such a case, a message is sent back to check for synonyms, and this adds extra latency and traffic. Finally, it also requires a directory at the LLC containing information about every block at any L1 cache, which is not always the case. In our implementation we analyze this optimization, as shown in Figure 2c.

6.3.2 VIPS-M operation

In contrast, VIPS-M does not need hardware synonym checking nor reverse translations, and this makes moving the TLB after the network straightforward (see Figure 3c). The benefits are substantial:

- we eliminate the TLB consistency problem,
- we increase the aggregate effective TLB capacity due to the lack of replicated entries,
- we can simplify private/shared classification because of the shared property of the TLB.
- at this position, the TLB can be manipulated by one of the general-purpose cores of the chip, freeing the simple cores of a GPU/accelerator from the burden of handling TLB misses or other memory management functions.

6.3.3 Banked TLB

A single TLB can restrict parallel access to the LLC (e.g., if the LLC is multibanked) and under heavy loads become a bottleneck. To eliminate this possibility a single logical TLB can be partitioned in the same way the LLC is distributed

Table 1: Base system parameters

Memory Parameters	
Processor frequency	3.0GHz
Block size	64 bytes
MSHR size/Delay timeout	16 entries/1000 cycles
Split L1 I & D caches	32KB, 4-way
L1 cache hit time	1 (tag) and 2 (tag+data) cycles
Shared unified LLC cache	8MB, 512KB/tile, 16-way
LLC bank cache hit time	6 (tag) and 12 (tag+data) cycles
L1-LLC inclusion policy	Inclusive
Directory	Full-map in LLC tags
Memory access time	160 cycles
Page size	4KB (64 blocks)
Split L1 TLB I & D	64 entries, fully associative
Unified L2 TLB	128 sets, 4 ways (512 entries)
TLB miss latency	800 cycles
Network Parameters	
Topology	2-dimensional mesh (4x4)
Routing technique	Deterministic X-Y
Flit size	16 bytes
Data message size	72 bytes (5 flits)
Control message size	8 bytes (1 flit)
Routing time	2 cycles
Switch time	2 cycles
Link time	2 cycles

(banked). However, TLBs are indexed by virtual page addresses, while LLCs are indexed by physical addresses. This may result in accessing a different TLB bank than its LLC counterpart. To avoid this situation, we force the LLC and the TLB to be in the same bank for every request. To guarantee this, LLC banks should be interleaved at least at a page-size granularity. In addition, the bits used to select the home bank should not change in the virtual-to-physical translation. Although, this re-introduces page coloring (to ensure that both the virtual and physical home bits are always the same), it can be an acceptable trade-off for the unprecedented simplification (simple virtual-cache coherence and elimination of TLB coherence) of a multicore memory system.

7. EVALUATION METHODOLOGY

We evaluate both a directory-based protocol (implementing MESI states) and a self-invalidation request-response protocol (VIPS-M) for the three different TLB placement designs described in the previous section. This leads to our six configuration evaluated: *MESI*-cT1nl, *VIPS-M*-cT1nl, *MESI*-c1Tnl, *VIPS-M*-c1Tnl, *MESI*-c1nTl, and *VIPS-M*-c1nTl.

For carrying out the evaluation we use the Simics full-system simulator [23] and the cycle-accurate GEMS simulator [24]. We also employ the GARNET network simulator [1] to model the interconnection network. Our target system is a 16-tile chip multiprocessor. Table 1 gives details about the main parameters of our base system. TLB miss latency considers invoking the OS routine and four accesses to the memory hierarchy to walk the page table (as in the 48-bit x86-64 machines). Both energy consumption and area requirements of the structures simulated have been calculated using the CACTI 6.5 tool [25] assuming a 32nm process technology.

We employ a wide variety of parallel applications (16) in our evaluations. *Barnes* (16K particles), *Cholesky* (tk16), *FFT* (64K complex doubles), *FMM* (16K particles), *LU* (512×512 matrix), *Ocean* (514×514 ocean), *Radiosity* (room, -ae 5000.0 -en 0.050 -bf 0.10), *Raytrace* (teapot, optimized version that removes unnecessary locks), *Volrend* (head), *Water-Nsq* (512 molecules) and *Water-Sp* (512 molecules) belong to the *SPLASH-2* benchmark suite [37]. *Em3d* (38400 nodes, 15% remote) is a shared-memory implementation of the Split-C benchmark. *Tomcatv* (256 points, 5 time steps) is a shared-memory implementation of the SPEC benchmark. *Blackscholes* (simmedium), *Swaptions* (simsmall), and *x264* (simsmall) are from the PARSEC benchmark suite [4]. We simulate the entire applications, but collect statistics only from start to completion of their parallel part.

8. RESULTS

8.1 TLB behavior

The placement of the TLB affects both the number of accesses and the number of misses. When the TLB is placed before the L1 cache (i.e., physical caches), every access causes a TLB translation. Figure 4 shows number of TLB accesses normalized with respect to a MESI protocol with physical caches (*MESI*-cT1nl). The number of TLB accesses is similar for the two protocols with physical caches (*MESI*-cT1nl and *VIPS-M*-cT1nl) since both require a TLB access before accessing the L1. However, when moving the TLB after the L1 cache (i.e., virtual caches) the number of TLB accesses drops considerably. Specifically, 98.8% of the accesses are eliminated. As we will see, this results in an important reduction in the energy consumed by the system.

While moving the TLB after the L1 does not significantly affect the number of TLB misses, moving it after the network has a substantial effect, as shown in Figure 5. This is because private TLBs replicate many page translations, specially when parallel applications are running in the multicore. Moving the TLB after the network and sharing the TLB, reduces the number of misses by 87%. This reduction translates into improvements in execution time as we show later.

8.2 Energy consumption

Figure 6 shows the energy consumed by the main structures in the memory hierarchy of the multicore. As we can see, the L1 cache is accounting for most of the energy consumed (around 70%, on average). Virtual caches require extra bits in the tag fields (process ID and permission bits), which makes their consumption even higher (around 73%, on average). However, the important reductions in the number of TLB accesses when employing virtual caches lead to significant savings in the total energy consumption, since the energy consumed by the TLB is around 20%, on average.

On the other hand, the energy consumed by the R-tag memory for a directory-based 16-core multicore is negligible, because the structure is small and is accessed only on misses. However, for other protocols such as those that are broadcast-based or for larger-scale multicores, where the number of invalidations can be higher, this consumption can increase and have some impact on the total consumption.

Finally, we can see energy reductions in VIPS-M compared to MESI in the network and in the LLC. Energy reductions in the network are due to sending back to the LLC only modified words. Energy savings in the LLC are mainly due to its smaller tag size, since VIPS-M does not include a full-map field per LLC entry. All these aspects lead to 19.5% energy savings for VIPS-M with virtual caches and shared TLBs (*VIPS-M*-c1nTl) when compared to a base MESI protocol with physical caches.

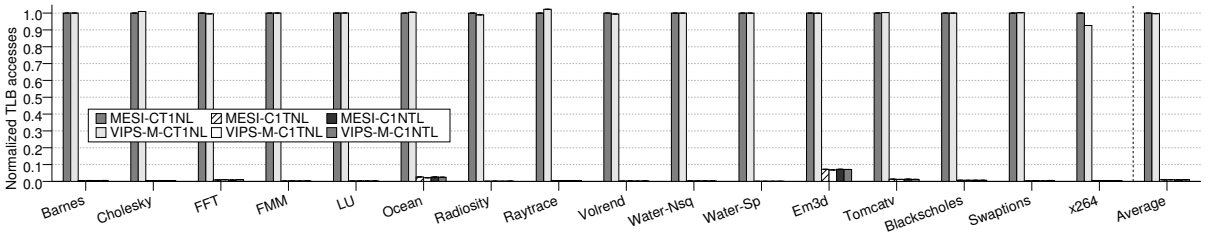


Figure 4: Number of TLB accesses for the configurations evaluated

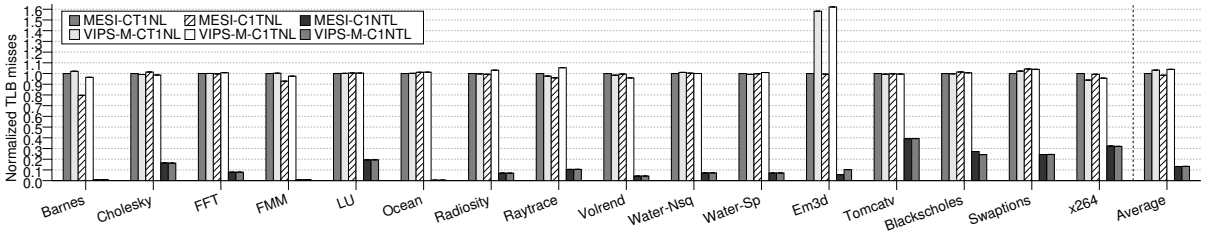


Figure 5: Number of TLB misses for the configurations evaluated

Table 2: Meta-data area requirements (mm^2)

Configuration	TLB	L1 tags	LLC tags	R-tag	Total	Meta (%)	Total (%)
MESI-CT1NL	0.4736	0.1152	1.1440	—	1.7328	—	—
VIPS-M-CT1NL	0.4736	0.1152	0.5456	—	1.1344	34.5%	2.0%
MESI-C1TNL	0.2720	0.1632	1.1440	0.1392	1.7184	0.8%	0.0%
VIPS-M-C1TNL	0.2720	0.1632	0.5456	—	0.9808	43.4%	2.6%
MESI-C1NTL	0.2720	0.1632	1.1440	0.1392	1.7184	0.8%	0.0%
VIPS-M-C1NTL	0.2720	0.1632	0.5456	—	0.9808	43.4%	2.6%

8.3 Execution time

Execution time is reduced when using shared TLBs, due to the reduction in the number of TLB misses. However, in MESI this option is not practical because of the need to check for synonyms. Since the address translation is not known until the network is crossed, synonym checking may require going back to the L1, which means extra network traffic and latency (even with the directory optimization described in § 6.3). This is why VIPS-M achieves better performance than MESI for the third configuration (Figure 7). In particular, an improvement in execution time by 5.4% is obtained for *VIPS-M*-C1NTL with respect to *MESI*-CT1NL (physical caches), while the corresponding MESI configuration (*MESI*-C1NTL) experiences a slowdown of 7.2%.

8.4 Area requirements and scalability

Table 2 shows the area requirements in mm^2 (as reported by CACTI 6.5 [25]) of the meta-data for each configuration evaluated (i.e., TLBs, L1 tags, LLC tags, and R-tag memory). The total and the reduction (in %) with respect to *MESI*-CT1NL is shown. Finally, the last column gives the reduction when considering also the data arrays of the L1s and the LLC.

Since the TLB is not in the critical path of the cache access when employing virtual caches, we can use a single-level TLB, instead of the two-level TLB required by physical caches (as previously indicated in Table 1). The elimination of the fully-associative instruction and data L1 TLBs brings important savings in area requirements.

On the other hand, the use of virtual caches requires the addition of extra bits to the tag field in order to store the process ID and the permission bits (6 and 3 bits, respectively). This increases the L1 tag area for the configurations employing virtual caches (C1TNL and C1NTL).

One of the main advantages of VIPS-M with respect to directory protocols in terms of area requirements is the removal of the directory (e.g., from the LLC tags). This makes VIPS-M also a more scalable solution, since the bits required by a bit-vector sharing field in a directory protocol grow linearly with the number of cores.

Finally, MESI directory protocols require one R-tag memory per virtual L1 cache. The information stored in this structure is the tag bits, a valid bit, and a 9-bit pointer that identifies the set and way where the block is stored in the virtual L1 cache.

On average, going from physical caches to virtual caches, the area of the meta-data in the system is reduced by around 43.4% with VIPS-M, while for MESI protocols the corresponding area reduction is only 0.8%. If we consider the total area of the memory hierarchy, the reduction obtained by VIPS-M is 2.6%.

8.5 TLB sensitivity analysis

As we have seen the C1NTL configuration, by virtue of its shared TLB that has no replication in its translations, outperforms the other configurations while at the same time reduces overall complexity by not requiring TLB coherence. Its outstanding TLB utilization, brings up the question of whether it can be traded-off for additional area savings without inordinately hurting performance. To this end, we performed a sensitivity analysis for *VIPS-M*-C1NTL, by reducing its TLB size to one-half and one-quarter. Our results show that we can halve the number of sets in the TLB without significantly degrading performance (0.2%, on average). Additionally, the performance degradation of having a four times smaller TLB is just 1.8% with respect to *VIPS-M*-C1TNL, while still being better than any MESI configuration. The area reduction compared to the baseline improves to 48.7% and 52.5% for one-half and one-quarter the TLB size, respectively.

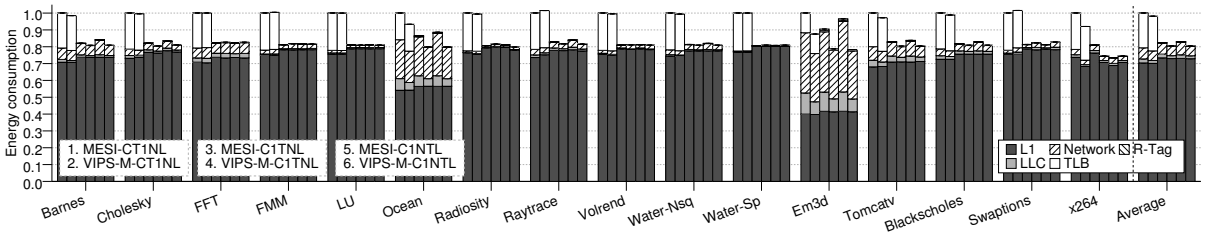


Figure 6: Dynamic energy consumed by the cache hierarchy and extra structures for the configurations evaluated

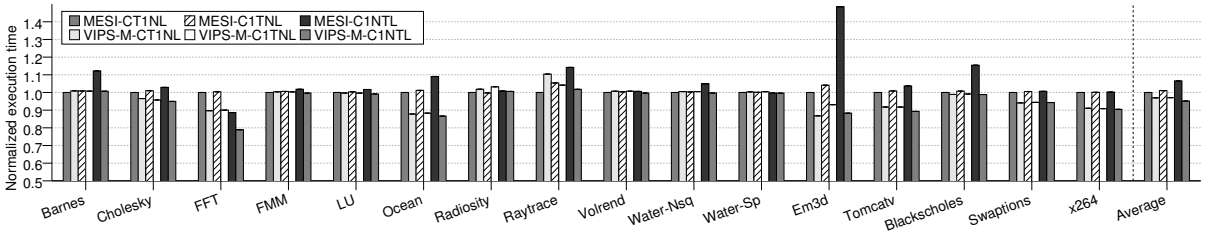


Figure 7: Normalized execution time for the configurations evaluated

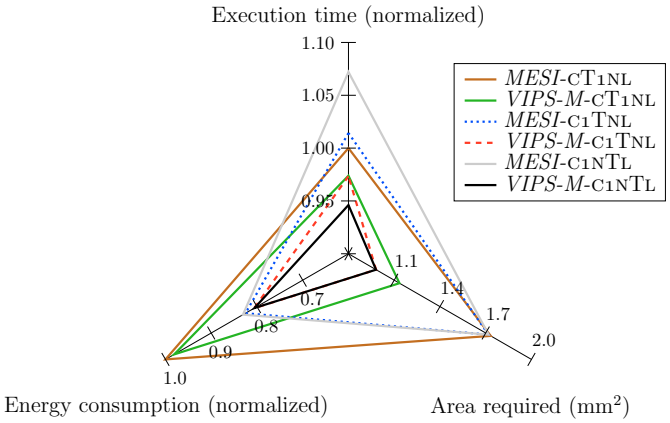


Figure 8: Time, energy, and area summary

8.6 Summary of results

Our six evaluated configurations (three TLB placement options × two protocols) affect differently our three metrics (execution time, energy, and area), making it hard to develop an overall view. We attempt to do so in Figure 8 where we plot in three dimensions (one metric per axis), the average results for all six configurations. Execution time and energy consumption are normalized to MESI-CT1NL, while area is in absolute values (mm^2). Figure 8 allows us to clearly see the trade-offs.

Physical caches vs. virtual caches:

- MESI configurations improve only energy, not execution time (worse), or area (same).
- VIPS-M configurations improve all three metrics while VIPS-M-C1NTL (shared TLB) improves only execution time over VIPS-M-C1TNL (private TLBs).

MESI vs. VIPS-M: For the same TLB placement design choice, the VIPS-M configuration improves area and execution time over the corresponding MESI configuration, but both have comparable energy consumption (with MESI having slightly worse). Finally, the winner in all three metrics is *VIPS-M*-C1NTL (improvement in area 43.4%, time 5.4%, and energy 20% with respect to *MESI*-CT1NL).

9. CONCLUSIONS

While the benefits of virtual caches are obvious for implementing coherent shared virtual memory in heterogeneous multicores, the complexity and cost of their coherency is a serious obstacle. We take a new perspective on this problem and show that it can be solved by re-thinking coherence. Our key observation is that a protocol without invalidations, downgrades, or forwardings, escapes the complexity of reverse translation that would be needed otherwise. By using such a protocol, not only we simplify virtual-cache coherence, but we simultaneously gain an increase in performance (5.4% over the base case of physical caches with a MESI-directory protocol), a reduction in energy (19.5% of the entire on-chip memory system and network), and a reduction in area (43.4% of the cache meta-data and address-translation structures). Further, based on our solution, we identify a new design point for TLB placement, a shared TLB before the LLC, which is particularly effective since it maximizes performance gains and energy and area savings, while at the same time also eliminates the complexity and overhead of TLB consistency.

10. ACKNOWLEDGMENTS

This work is supported, in part, by the Swedish Research Council UPMARC Linnaeus Centre, the Spanish MINECO under grant TIN2012-38341-C04-03, and the EU Projects HEAP FP7-ICT-247615 and LPGPU FP7-ICT-288653.

11. REFERENCES

[1] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. GARNET: A detailed on-chip network model inside a full-system simulator. In *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 33–42, Apr. 2009.

[2] A. Basu, M. D. Hill, and M. M. Swift. Reducing memory reference energy with opportunistic virtual caching. In *39th*

Int'l Symp. on Computer Architecture (ISCA), pages 297–308, June 2012.

[3] A. Bhattacharjee, D. Lustig, and M. Martonosi. Shared last-level tlbs for chip multiprocessors. In *17th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 62–73, Feb. 2011.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, Oct. 2008.

[5] D. L. Black, R. F. Rashid, D. B. Golub, C. R. Hill, and R. V. Baron. Translation lookaside buffer consistency: A software approach. In *3th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 113–122, Apr. 1989.

[6] P. Boudier and G. Sellers. Memory system on fusion APUs: The benefits of zero copy. In *AMD Fusion developer summit*, June 2011.

[7] M. Cekleov and M. Dubois. Virtual-address caches part 1: Problems and solutions in uniprocessors. 17(5):64–71, Sept. 1997.

[8] M. Cekleov and M. Dubois. Virtual-address caches, part 2: Multiprocessor issues. 17(6):69–74, 1997.

[9] C. Chao, M. Mackey, and B. Sears. Mach on a virtually addressed cache architecture. Technical Report HPL-90-67, HP Laboratories, June 1990.

[10] D. R. Cheriton, G. A. Slavenburg, and P. D. Boyle. Software-controlled caches in the vmp multiprocessor. In *13th Int'l Symp. on Computer Architecture (ISCA)*, pages 366–374, June 1986.

[11] B. Choi, R. Komuravelli, H. Sung, et al. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *20th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 155–166, Oct. 2011.

[12] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *38th Int'l Symp. on Computer Architecture (ISCA)*, pages 93–103, June 2011.

[13] E. Demers. Summit keynote: Evolution of amd's graphics core, and preview of graphics core next. In *AMD Fusion developer summit*, June 2011.

[14] J. R. Goodman. Coherency for multiprocessor virtual address caches. In *2th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 72–81, Apr. 1987.

[15] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *36th Int'l Symp. on Computer Architecture (ISCA)*, pages 184–195, June 2009.

[16] M. D. Hill, S. J. Eggers, J. R. Larus, et al. SPUR: A VLSI multiprocessor workstation. Technical Report UCB/CSD-86-273, EECS Department, University of California, Berkeley, Dec. 1985.

[17] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative shared memory: Software and hardware for scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 11(4):300–318, Nov. 1993.

[18] N. P. Jouppi. Architectural and organizational tradeoffs in the design of the MultiTitan CPU. In *16th Int'l Symp. on Computer Architecture (ISCA)*, pages 281–289, June 1989.

[19] S. Kaxiras and G. Keramidas. SARC coherence: Scaling directory cache coherence in performance and power. *IEEE Micro*, 30(5):54–65, Sept. 2011.

[20] D. Kim, J. A. J. Kim, and J. Huh. Subspace snooping: Filtering snoops with operating system support. In *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122, Sept. 2010.

[21] L. Kohn and N. Margulis. Introducing the intel i860 64-bit microprocessor. *IEEE Micro*, 9(4):15–30, July 1989.

[22] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *22nd Int'l Symp. on Computer Architecture (ISCA)*, pages 48–59, June 1995.

[23] P. S. Magnusson, M. Christensson, J. Eskilson, et al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[24] M. M. Martin, D. J. Sorin, B. M. Beckmann, et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 33(4):92–99, Sept. 2005.

[25] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Cacti 6.0. Technical Report HPL-2009-85, HP Labs, Apr. 2009.

[26] J. M. Owen, M. D. Hummel, D. R. Meyer, and J. B. Keller. System and method of maintaining coherency in a distributed communication system. U.S. Patent 7069361, June 2006.

[27] X. Qiu and M. Dubois. Tolerating late memory traps in ilp processors. In *26th Int'l Symp. on Computer Architecture (ISCA)*, pages 76–87, May 1999.

[28] X. Qiu and M. Dubois. The synonym lookaside buffer: A solution to the synonym problem in virtual caches. *IEEE Transactions on Computers (TC)*, 57(12):1585–1599, Dec. 2008.

[29] A. Ros and S. Kaxiras. Complexity-effective multicore coherence. In *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 241–252, Sept. 2012.

[30] I. Singh, A. Shriraman, and W. W. L. Fung. Cache coherence for gpu architectures. In *19th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 578–590, Feb. 2013.

[31] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*, volume 6 of *Synthesis Lectures on Computer Architecture*. Morgan & Claypool Publishers, May 2011.

[32] P. J. Teller. Translation-lookaside buffer consistency. *IEEE Computer*, 23(6):26–36, June 1990.

[33] P. H. Wang, J. D. Collins, G. N. Chinya, et al. Exochi: Architecture and programming environment for a heterogeneous multi-core multithreaded system. In *2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 156–166, June 2007.

[34] W.-H. Wang, J.-L. Baer, and H. M. Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *16th Int'l Symp. on Computer Architecture (ISCA)*, pages 140–148, June 1989.

[35] B. Wheeler and B. N. Bershad. Consistency management for virtually indexed caches. In *5th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 124–136, Oct. 1992.

[36] H. Wong, A. Bracy, E. Schuchman, et al. Pangaea: A tightly-coupled ia32 heterogeneous chip multiprocessor. In *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 52–61, Oct. 2008.

[37] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Int'l Symp. on Computer Architecture (ISCA)*, pages 24–36, June 1995.

[38] D. A. Wood, S. Chandra, B. Falsafi, et al. Mechanisms for cooperative shared memory. In *20st Int'l Symp. on Computer Architecture (ISCA)*, pages 156–167, May 1993.

[39] L. Zhang, E. Speight, R. Rajamony, and J. Lin. Enigma: Architectural and operating system support for reducing the impact of address translation. In *24th Int'l Conf. on Supercomputing (ICS)*, pages 159–168, June 2010.